



Singularity

**James Larus
Galen Hunt
Ben Zorn**

**MSR TAB Presentation
June 28, 2004**



Singularity

**James Larus
Galen Hunt
Ben Zorn**

**MSR TAB Presentation
June 28, 2004**



Singularity

**James Larus
Galen Hunt
Ben Zorn**

**MSR TAB Presentation
June 28, 2004**



Dependable Artifacts

- Characteristics
 - utility
 - usability
 - predictability
 - reliability
 - cost



Dependable Artifacts

- Characteristics
 - utility
 - usability
 - predictability
 - reliability
 - cost





Dependable Artifacts

- Characteristics

- utility
- usability
- predictability
- reliability
- cost





Software is Not Dependable

- For most people, software
 - behaves unpredictably and inexplicably
 - fails often
 - feels insecure, corruptible
 - appears unreliable and shoddy
- Unhappy state of affairs
 - novelty is wearing off
 - day-to-day tool, not new toy





Why Software is Not Dependable

- Good design is difficult
 - no proven, teachable methodology
- Producing error-free code is hard
 - PPRC's focus
- Minor mistakes have major consequences
 - system architecture is not resilient
- Testing is not systematic
 - but no other general alternative to reduce errors
- Culture emphasizes measurable aspects
 - performance
 - features



Improving Software Technology

- Fundamental technologies
 - programming languages
 - programming tools
 - system architecture and programming model (OS/RT)
- Steady, incremental improvement in each field
- Major improvements cross discipline boundaries
 - legacy and narrow focus constrain progress in each area
- Example: correctness tools
 - languages difficult to analyze soundly
 - specifications not part of code
 - programs run in open and porous environments



Singularity

- Could we build better software if we start afresh?
- Can we build systems that are resilient against programmer faults?



Singularity

- Could we build better software if we start afresh?
- Can we build systems that are resilient against programmer faults?
- How would software development evolve if dependability was primary goal?



Singularity

- Could we build better software if we start afresh?
- Can we build systems that are resilient against programmer faults?
- How would software development evolve if dependability was primary goal?
- Can we accelerate progress in software by attacking problems from several directions?



Singularity

- Could we build better software if we start afresh?
- Can we build systems that are resilient against programmer faults?
- How would software development evolve if dependability was primary goal?
- Can we accelerate progress in software by attacking problems from several directions?
- What technology will MS need to build dependable software?



Singularity Big Bets

- Use managed code throughout to reduce errors and virtualize instruction stream
- Constrain language and programming model to favor verification
- Close execution environment to facilitate compilation and program analysis
- Introduce strong process isolation to minimize consequence of programmer faults
- Make communication explicit and carefully checked



Overview

- **Singularity OS (Hunt)**
- Programming Language and Tools (Larus)
- Compiler and Runtime Environment (Zorn)
- Application Domain & Validation
- Conclusion



OS Research Big Bets



OS Research Big Bets

- The Singularity razor
 - no unsafe code
 - no dynamic loading of code into existing processes
 - no cross-process communication through shared memory
 - no untyped communication
 - no code w/o metadata



OS Research Big Bets

- **The Singularity razor**
 - no unsafe code
 - no dynamic loading of code into existing processes
 - no cross-process communication through shared memory
 - no untyped communication
 - no code w/o metadata
- **Benefits**
 - improved verification and analysis for code reliability
 - strong fault containment for application reliability
 - strong application specification for system reliability
 - fixed process identity for security
 - optimization of whole processes for performance



OS Research Big Bets

- **The Singularity razor**
 - no unsafe code
 - no dynamic loading of code into existing processes
 - no cross-process communication through shared memory
 - no untyped communication
 - no code w/o metadata
- **Benefits**
 - improved verification and analysis for code reliability
 - strong fault containment for application reliability
 - strong application specification for system reliability
 - fixed process identity for security
 - optimization of whole processes for performance
- **Research opportunities**
 - performance and programmability
 - multi-resource real-time resource management and real time GC
 - process-based recovery from programmer failure
 - making system configuration and construction declarative

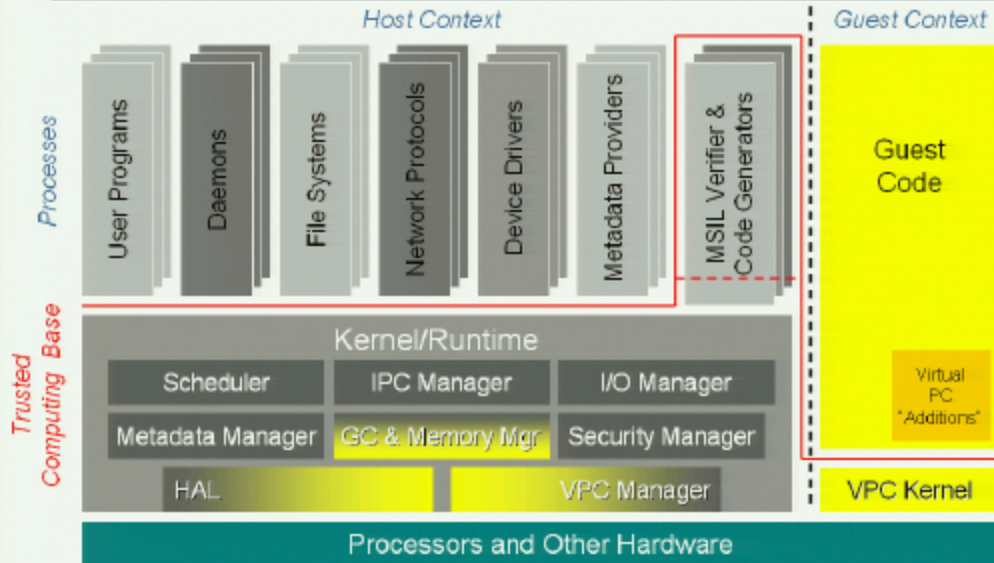


Adding Isolation to Managed Code

- Managed code doesn't provide isolation or fault containment
- Strong process isolation model
 - no shared memory between processes
 - no dynamic code loading, process sealed at creation
- Singularity process
 - isolated object space with threads
 - unit of failure and termination
 - unit of security policy enforcement
 - unit of system extension
 - target for complete optimization
- Is not
 - address space
 - extensible once started



Singularity OS Architecture



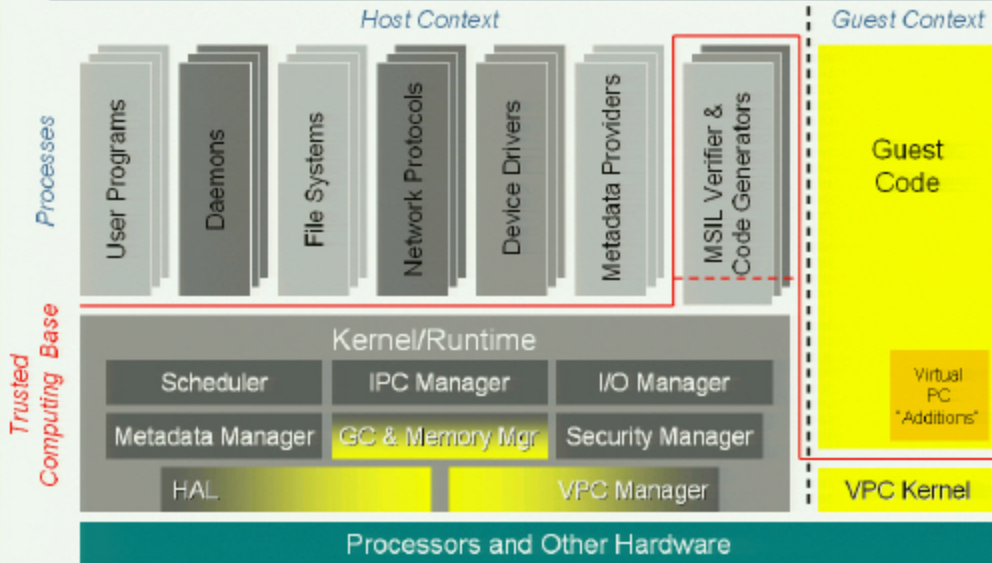


Adding Isolation to Managed Code

- Managed code doesn't provide isolation or fault containment
- Strong process isolation model
 - no shared memory between processes
 - no dynamic code loading, process sealed at creation
- Singularity process
 - isolated object space with threads
 - unit of failure and termination
 - unit of security policy enforcement
 - unit of system extension
 - target for complete optimization
- Is not
 - address space
 - extensible once started



Singularity OS Architecture



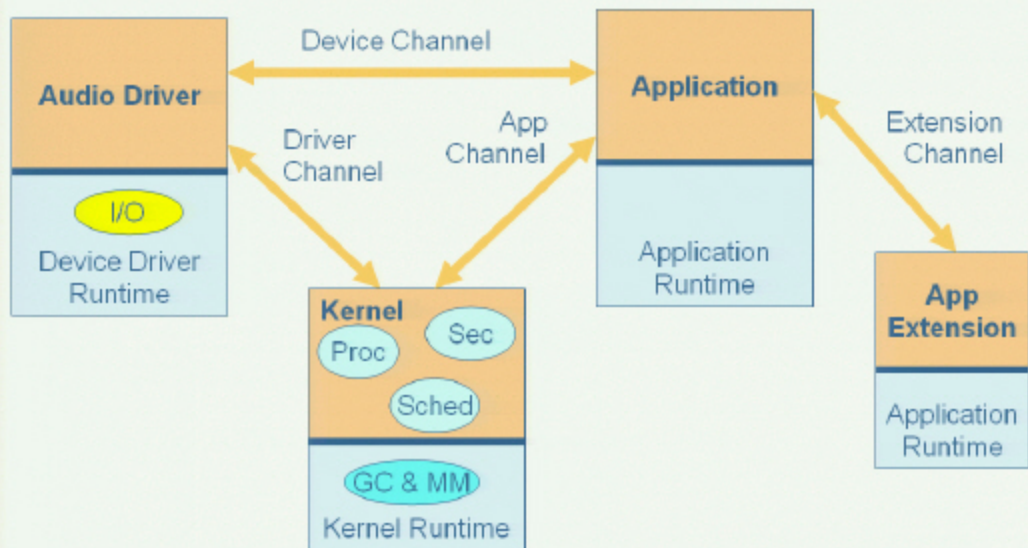


Communication

- Explicit communication model
 - all communication through channels typed by contract
 - contract defines types and patterns of messages
- Singularity channel
 - bi-directional, asynchronous message transport
 - accessed through exactly two endpoints
 - moved by placing endpoint in a message
- Is not
 - movable except in statically verified state
 - accessible after endpoint has moved



Processes





Administration and Maintenance

- Ubiquitous metadata
 - administration policies described in declarative metadata
 - setup and patching are OS, not application, responsibility
- Singularity application abstraction
 - first-class OS abstraction describing an application
 - statically described in an application manifest
 - complete descriptions of entire system (including kernel)
 - basis for installation and maintenance
 - dynamically exposed for introspection



Security

- Orthogonal administration of isolated applications
 - application isolation instead of virtual PCs
- Singularity process-based security model
 - fixes identity at creation time [code,user,machine]
 - arbitrates process creation
 - arbitrates channel endpoint propagation
 - supports absolute process content privacy
- Model does not
 - use CLR code access security
 - provide security granularity smaller than process



Overview

- Singularity OS (Hunt)
- **Programming Language and Tools (Larus)**
- Compiler and Runtime Environment (Zorn)
- Application Domain & Validation
- Conclusion



Language Starting Points

- Spec#
 - superset of C# developed by FSE and SPT groups
 - pre-/post-conditions & invariants
 - statically verify subset
 - runtime assertions and test case generation
 - vehicle for language experimentation
 - evolve language with tools
- Conformance checking
 - behaviorally typed message passing channels
 - ensure client/server code obeys channel protocol
 - modular checks lead to global deadlock freedom



Language Research Goals

- Language to support reliable software
 - focus on problematic aspects of programming
 - balance static verification and runtime mechanisms
 - completeness/precision
 - expressiveness/late notification
- Test new features against realistic, but manageable code base
- Basis for Java/C# successor



Language Research Topics

- Error handling
 - lightweight transactions to restore state
 - static checking of explicit restore code
- Concurrency
 - atomic block identifies unit of concurrent update
 - enhanced types identify shared data
 - optimistic & pessimistic implementations
- Modularity
 - functions & classes are mechanism for building abstractions
 - need better large-grain mechanisms (modules)



Tools Starting Points

- Build on research in SPT, FSE, and TVM groups
- Boogie extended static checker for Spec#
 - sound
 - object invariants
 - annotation inference
- Zing model checker
 - targeted at language verification
 - used in a variety of concurrency-related tools
- Test generation tools driven from Spec# annotations
- Managed code tools from PPRC



Language Research Topics

- Error handling
 - lightweight transactions to restore state
 - static checking of explicit restore code
- Concurrency
 - atomic block identifies unit of concurrent update
 - enhanced types identify shared data
 - optimistic & pessimistic implementations
- Modularity
 - functions & classes are mechanism for building abstractions
 - need better large-grain mechanisms (modules)



Tools Starting Points

- Build on research in SPT, FSE, and TVM groups
- Boogie extended static checker for Spec#
 - sound
 - object invariants
 - annotation inference
- Zing model checker
 - targeted at language verification
 - used in a variety of concurrency-related tools
- Test generation tools driven from Spec# annotations
- Managed code tools from PPRC



Tools Research Goals

- Accelerate co-evolution of languages and tools
- Employ tools throughout entire project lifetime
- Scale tools from components (e.g. drivers) to systems
- Identify opportunities for new tools and build them



Tools Research Topics

- Formal design
 - modeling tools for critical aspects of system
 - formal specification, simulation, and verification
- Correctness
 - continue research
- Security & privacy
 - beyond exploits to system-level problems
 - information flow
 - runtime monitoring
- Testing
 - static analysis to increase test effectiveness
 - usage profiles



Overview

- Singularity OS (Hunt)
- Programming Language and Tools (Larus)
- **Compiler and Runtime Environment (Zorn)**
- Application Domain & Validation
- Conclusion



Compiler and Managed Runtime Environment

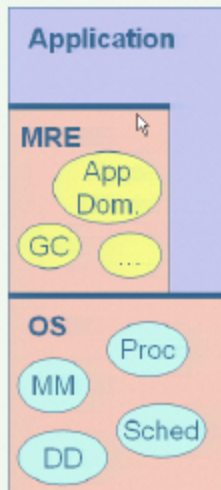
- Big bets
 - merging OS and managed code runtime
 - limits on functionality benefit performance and correctness
 - e.g., no dynamic code loading
 - real-time and kernel garbage collection
- Managed Runtime Environment (MRE)
 - IL: architecture neutral instruction set
 - base class library: collections, I/O, etc.
 - services: garbage collection, reflection, ...
- Existing MREs designed for applications, not system





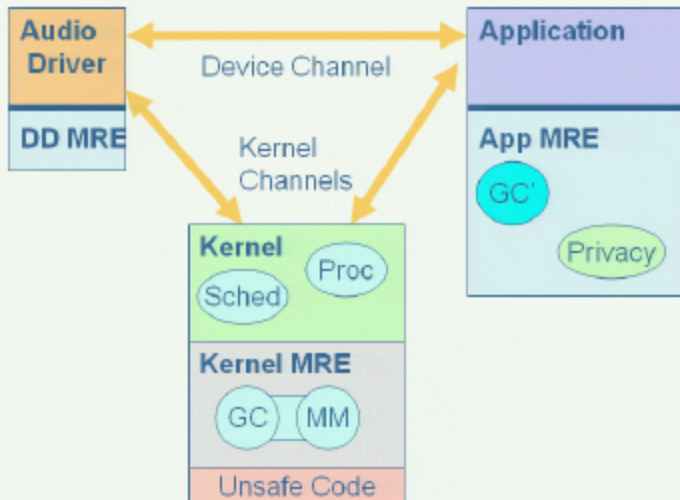
OS / MRE Integration

Current Systems



Singularity

Singularity



Microsoft Confidential

25



Research Goals

- **Compiler**
 - produce high performance managed system
 - integrate types into lowest levels of system
- **Runtime**
 - make GC suitable for kernel
 - exploit runtime to improve correctness and performance
 - rethink runtime services (e.g., reflection)
 - customize MREs for application domains



Compiler Research Topics

- Typed assembly language (TAL)
 - create a TAL suitable for OO languages
 - also suitable for MRE/OS low-level implementation (GC)
 - show MRE + application program is still type safe
 - e.g.: program type safe after inlining runtime type test
- Optimization
 - whole program: take advantage of process sealing
 - cross-process (system-level) optimization
 - e.g.: channel communication
- New language features
 - e.g.: transactions



Runtime Research Topics

- GC in kernel
 - exploit MM info to increase GC locality
 - integrate GC with scheduling for real-time apps
 - support multiple GCs for diverse app requirements
- Use runtime to enhance correctness
 - GC has global view of heap connectivity
 - sampling can collect arbitrary runtime info
- Rethinking runtime
 - move reflection to earlier time (e.g., loading)
 - defining MREs as unit of abstraction



Compilation / MRE Starting Points

- Build on research in ACT and RAD groups
- First compiler / MRE based on Bartok
 - optimizing MSIL to native code compiler
 - lightweight runtime system
 - runtime and compiler written entirely in C#
- Ongoing runtime locality and correctness research
 - hot data streams
 - statistical error detection
 - runtime data collection and sampling



Overview

- Singularity OS (Hunt)
- Programming Language and Tools (Larus)
- Compiler and Runtime Environment (Zorn)
- **Application Domain & Validation**
- **Conclusion**



Application: Home Server

- Platform for home services
 - multiple, interacting applications
 - zero maintenance / remotely administered
 - secure
 - Robust / recover and continue after failure
- Home services
 - video/audio server (TiVo)
 - backup (Miva)
 - home automation
 - communication/collaboration (Exchange)
- Alternative to single-function appliances
 - lower fixed cost
 - better integration
 - lower barrier for innovation (Wang vs PC)
- Infrastructure for software as service
- TAB suggestions are very welcomed (want a device for home?)



Participants

Advanced Compiler Technology

Bjarne Steensgaard
David Tarditi
Juan Chen
Pramod Joisha

Distributed Systems (Cambridge)

Marc Shapiro

Foundation of SW Engineering

Nikolai Tillmann
Wolfgang Grieskamp
Wolfram Schulte

MSR Silicon Valley

Martin Abadi
Roy Levin
Ted Wobber
Ulfar Erlingsson

Performance Monitoring and Analysis

Ben Zorn

Software Productivity Tools

Jakob Rehof
Jim Larus
Manuel Fahndrich
Sriram Rajamani

Systems and Networking

Bill Bolosky
Brian Zill
Dan Simon
Galen Hunt
Mike Jones
Steven Levi

Systems and Performance (Cambridge)

Andrew Herbert
Austin Donnelly
David Stewart
Dushyanth Narayanan
Neil Stratford
Paul Barham
Rebecca Isaacs
Richard Black
Richard Mortier

Testing, Verification, And Measurement

Madan Musuvathi
Tom Ball

Runtime Analysis and Design

Trishul Chilimbi



Status and Milestones

- June 2004:
 - simple Singularity kernel with threads and channels
 - Bartok runtime system on bare metal
 - type safe device drivers and PnP support
 - Spec# compiler generating MSIL
- September 2004
 - Spec# extensions
 - real-time scheduler
 - more device drivers
 - services (name service, file system)
 - processes and more efficient threads and channels
 - application abstraction (metadata)
- December 2004
 - first application deployment to home machines



Conclusion

- Imperative to develop tools and techniques to build dependable software
- Dependability requires major, synergistic changes in languages, tools, and system architecture
- Singularity deliverables:
 1. specification and modeling tools
 2. verifiable programming languages
 3. program correctness tools
 4. real-time managed code system
 5. strong isolation OS
 6. self-descriptive system and applications



Backup Slides